Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions

Semih Okur[1], Cansu Erdogan[1], and Danny Dig[2]

[1] University of Illinois at Urbana-Champaign, USA,
{okur2, cerdoga2}@illinois.edu
[2] Oregon State University, USA, digd@eecs.oregonstate.edu

Abstract. Parallel libraries continuously evolve from low-level to higher-level abstractions. However, developers are not up-to-date with these higher-level abstractions, thus their parallel code might be hard to read, slow, and unscalable. Using a corpus of 880 open-source C# applications, we found that developers still use the old `Thread` and `ThreadPool` abstractions in 62% of the cases when they use parallel abstractions. Converting code to higher-level abstractions is (i) tedious and (ii) error-prone. e.g., it can harm performance and silence the uncaught exceptions.
We present two automated migration tools, Taskifier and Simplifier that work for C# code. The first tool transforms old style `Thread` and `ThreadPool` abstractions to `Task` abstractions. The second tool transforms code with `Task` abstractions into higher-level design patterns. Using our code corpus, we have applied these tools 3026 and 405 times, respectively. Our empirical evaluation shows that the tools (i) are highly applicable, (ii) reduce the code bloat, (iii) are much safer than manual transformations. We submitted 66 patches generated by our tools, and the open-source developers accepted 53.

1 Introduction

In the quest to support programmers with faster, more scalable, and readable code, parallel libraries continuously evolve from low-level to higher-level abstractions. For example, Java 6 (2006) improved the performance and scalability of its concurrent collections (e.g., `ConcurrentHashMap`), Java 7 (2011) added higher-level abstractions such as lightweight tasks, Java 8 (2014) added lambda expressions that dramatically improve the readability of parallel code. Similarly, in the C# ecosystem, .NET 1.0 (2002) supported a Threading library, .NET 4.0 (2010) added lightweight tasks, declarative parallel queries, and concurrent collections, .NET 4.5 (2012) added reactive asynchronous operations.

Low-level abstractions, such as `Thread`, make parallel code more complex, less scalable, and slower. Because `Thread` represents an actual OS-level thread, developers need to take into account the hardware (e.g., the number of cores) while coding. Threads are *heavyweight*: each OS thread consumes a non-trivial amount of memory, and starting and cleaning up after a retired thread takes hundreds of thousands of CPU cycles. Even though a .NET developer can use

`ThreadPool` to amortize the cost of creating and recycling threads, she cannot control the behavior of the computation on `ThreadPool`. Moreover, new platforms such as Microsoft Surface Tablet no longer support `Thread`. .NET also does not allow using the new features (e.g., `async/await` abstractions) with `Thread` and `ThreadPool`. Furthermore, when developers mix old and new parallel abstractions in their code, it makes it hard to reason about the code because all these abstractions have different scheduling rules.

Higher-level abstractions such as .NET `Task`, a unit of parallel work, make the code less complex. `Task` gives advanced control to the developer (e.g., chaining, cancellation, futures, callbacks), and is more scalable than `Thread`. Unlike threads, tasks are *lightweight*: they have a much smaller performance overhead and the runtime system automatically balances the workload. Microsoft now encourages developers to use `Task` in order to write scalable, hardware independent, fast, and readable parallel code [26].

However, most developers are oblivious to the benefits brought by the higher-level parallel abstractions. In recent empirical studies for C# [18] and Java [25], researchers found that `Thread` is still the primary choice for most developers. In this paper we find similar evidence. Our corpus of the most popular and active 880 C# applications on Github [12] that we prepared for this paper, shows that when developers use parallel abstractions they still use the old `Thread` and `ThreadPool` 62% of the time, despite the availability of better options. Therefore, a lot of code needs to be migrated from low-level parallel abstractions to their higher-level equivalents.

The migration has several challenges. First, developers need to be aware of the different nature of the computation. While blocking operations (e.g., I/O operations, `Thread.Sleep`) do not cause a problem in Thread-based code, they can cause a serious performance issue (called thread-starvation) in Task-based code. Because the developers need to search for such operations deep in the call graph of the concurrent abstraction, it is easy to overlook them. For example, in our corpus of 880 C# applications, we found that 32% of tasks have at least one I/O blocking operation and 9% use `Thread.Sleep` that blocks the thread longer than 1 sec. Second, developers need to be aware of differences in handling exceptions, otherwise exceptions become ineffective or can get lost.

In this paper, we present an automated migration tool, Taskifier, that transforms old style `Thread` and `ThreadPool` abstractions to higher-level `Task` abstractions in C# code. During the migration, Taskifier automatically addresses the non-trivial challenges such as transforming blocking to non-blocking operations, and preserving the exception-handling behavior.

The recent versions of parallel libraries provide even higher-level abstractions on top of Tasks. For example, the `Parallel` abstraction in C# supports parallel programming design patterns: data parallelism in the form of parallel loops, and fork-join task parallelism in the form of parallel tasks co-invoked in parallel. These dramatically improve the readability of the parallel code. Consider the example in Code listing 1.1, taken from ravendb [1] application. Code listing 1.2 represents the same code with a `Parallel` operation, which dramatically reduces

the code. According to a study [15] by Microsoft, these patterns may also lead to better performance than when using `Task`, especially when there is a large number of work items (`Parallel` reuses tasks at runtime to eliminate the overhead).

Code 1.1 Forking Task in a loop

```
1  List<Task> tasks = new List<Task>();
2  for (int i = 0; i <= n; i++)
3  {
4    int copy = i;
5    Task taskHandle = new Task(
6           () => DoInsert(..., copy));
7    taskHandle.Start();
8    tasks.Add(taskHandle);
9  }
10 Task.WaitAll(tasks);
```

Code 1.2 Equivalent Parallel.For

```
1 Parallel.For(0,n,(i)=>DoInsert(...,i));
```

Despite the advantages of the higher-level abstractions in the `Parallel` class, developers rarely use them. In our corpus we found that only 6% of the applications use the `Parallel` operations. We contacted the developers of 10 applications which heavily use `Thread`, `ThreadPool`, and `Task` abstractions, and asked why they are not using the `Parallel` operations. The major reason given by developers was lack of awareness. This indicates there is a need for tools that suggest transformations, thus educating developers about better coding practices.

Transforming the Task-related code into higher-level `Parallel` operations is not trivial: it requires control- and data-flow analysis, as well as loop-carried dependence analysis. For the example in Listing 1.1, the code does not execute the assignment in Line 4 in parallel with itself in other iterations (only the code in the task body – Line 6 – is executed in parallel). However, after converting the original `for` into a `Parallel.For`, the assignment in Line 4 will also execute in parallel with other assignments. Thus, the programmer must reason about the loop-carried dependences.

Inspired from the problems that developers face in practice, we designed and implemented a novel tool, Simplifier, that extracts and converts Task-related code snippets into higher-level parallel patterns. To overcome the lack of developer awareness, Simplifier operates in a mode where it suggests transformations as "quick-hints" in the Visual Studio IDE. If the developer agrees with the suggestion, Simplifier automatically transforms the code.

This paper makes the following contributions:

Problem: To the best of our knowledge, this is the first paper that describes the novel problem of migrating low-level parallel abstractions into their high-level counterparts. We show that this problem appears in real-life applications by bringing evidence of its existence from a corpus of 880 C# open-source applications.

Algorithms: We describe the analysis and transformation algorithms which address the challenges of (i) migrating `Thread`-code into `Task` abstractions and (ii) transforming `Task` code snippets into higher-level `Parallel` design patterns.

Tools: We implemented our algorithms into two tools, Taskifier and Simplifier. We implemented them as extensions to Visual Studio, the primary development environment for C#.

Evaluation: We empirically evaluated our implementations by using our code corpus of 880 C# applications. We applied Taskifier 3026 times and Simplifier 405 times. First, the results show that the tools are widely *applicable*: Taskifier successfully migrated 87% of `Thread` and `ThreadPool` abstractions to `Task`. Simplifier successfully transformed 94% of suggested snippets to `Parallel`. Second, these transformations are *valuable*: Taskifier reduces the size of the converted code snippets by 2617 SLOC and Simplifier reduces by 2420 SLOC in total. Third, the tools save the programmer from manually changing 10991 SLOC for the migration to `Task` and 7510 SLOC for the migration to `Parallel`. Fourth, automated transformations are safer. Several of the manually written `Task`-based codes by open-source developers contain problems: 32% are using blocking operations in the body of the `Task`, which can result in thread-starvation. Fifth, open-source developers found our transformations useful. We submitted 66 patches generated by our tools and the open-source developers accepted 53.

2 Background on Parallel abstractions in .NET

Our tools target the parallelism paradigms in .NET. Here we give a gentle introduction to parallel programming in .NET. There are four main abstractions that allow developers to spawn asynchronous computation.

2.1 Thread

Operating systems use processes to separate the different applications that they are executing. Thread is the basic unit to which an operating system allocates processor time, and more than one thread can be executing code inside one process. Threading library in .NET provides an abstraction of threads, `Thread` class since its first version, 2003.

`Thread` represents an actual OS-level thread, so it is expensive to use; creating a `Thread` needs about 1.5 MB memory space. Windows also creates many additional data structures to work with this thread, such as a Thread Environment Block (TEB), a user mode stack, and a kernel mode stack. Bringing in new `Thread` may also mean more thread context switching, which further hurts performance. It takes about 200,000 CPU cycles to create a new thread, and about 100,000 cycles to retire a thread.

On one hand, `Thread` class allows the highest degree of control; developers can set many thread-level properties like the stack size, priority, background and foreground. However, general-purpose apps do not need most of these low-level features. On that matter, Microsoft discourages developers to use these features because they are usually misused [26]. In modern C# code, developers should rarely need to explicitly start their own thread.

On the other hand, `Thread` has some limitations. For example, a `Thread` constructor can take at most one parameter and this parameter must be of type `Object`. In Code listing 1.3, a `Thread` is first created with its body which is `MailSlotChecker` method. `ParameterizedThreadStart` indicates that this method

needs to take a parameter. After priority and background properties are set, the parameter, `info` is created and given to `Start` method that asynchronously executes the `Thread`. When the instance `info` of `MailSlotThreadInfo` type is passed to `Thread` body, it will be forced to upcast to `Object` type. Developers manually need to downcast it to `MailSlotThreadInfo` type in `MailSlotChecker` method. Hence, this introduced verbose code like explicit casting, `ParameterizedThreadStart` objects. To wait for the termination of the `Thread`, the code invokes a blocking method, `Join`.

Code 1.3 Thread usage example from Tiraggo [7] app

```
Thread thread = new Thread(new ParameterizedThreadStart(MailSlotChecker));
thread.Priority = ThreadPriority.Lowest;
thread.IsBackground = true;
MailSlotThreadInfo info = new MailSlotThreadInfo(channelName, thread);
thread.Start(info);
...
thread.Join(info);
```

2.2 ThreadPool

To amortize the cost of creating and destroying threads, a pool of threads can be used to execute work items. There is no need to create or destroy threads for each work item; the threads are recycled in the pool. .NET provides an abstraction, the `ThreadPool` class, since its first version.

Although `ThreadPool` class is efficient to encapsulate concurrent computation, it gives developers no control at all. Developers only submit work which will execute at some point. The only thing they can control about the pool is its size. `ThreadPool` offers no way to find out when a work item has been completed (unlike `Thread.Join()`), neither a way to get the result.

Code listing 1.4 shows two main examples of `ThreadPool` usage. `QueueUserWorkItem` is used to put work items to the thread pool. The first example executes `foo(param)` method call in the thread pool but it is unclear because of the syntax. The second example executes the same thing with a lambda function which is introduced in C# 4.0. Developers can directly pass the parameters to the lambda function. However, `QueueUserWorkItem` only accepts a lambda function that takes one parameter: `(x)=> ...`. Developers always need to provide one parameter, regardless of whether they use it or not, thus many times they call this parameter `unused` or `ignored`.

Code 1.4 ThreadPool example

```
1 ThreadPool.QueueUserWorkItem(new WaitCallback(foo),param);
2 ThreadPool.QueueUserWorkItem((unused)=> foo(param));
```

2.3 Task

The `Task` abstraction was introduced in the Task Parallel Library [16] with the release of .NET 4.0 in 2010. `Task` offers the best of both worlds, `Thread` and `ThreadPool`. `Task` is simply a lightweight thread-like entity that encapsulates an asynchronous operation. Like `ThreadPool`, a `Task` does not create its

own OS thread so it does not have high-overhead of `Thread`. Instead, it is executed by a `TaskScheduler`; the default scheduler simply runs on the thread pool. `TaskScheduler` use work-stealing techniques which are inspired by the Java fork-join framework [14].

Unlike the `ThreadPool`, `Task` also allows developers to find out when it finishes, and (via the generic `Task<T>`) to return a result. A developer can call `ContinueWith()` on an existing Task to make it run more code once the task finishes; if it's already finished, it will run the callback immediately. A developer can also synchronously wait for a task to finish by calling `Wait()` (or, for a generic task, by getting the Result property). Like `Thread.Join()`, this will block the calling thread until the task finishes.

The bottom line is that `Task` is almost always the best option; it provides a much more powerful API and avoids wasting OS threads. All newer high-level concurrency APIs, including PLINQ, `async/await` language features, and modern asynchronous methods are all built on `Task`. It is becoming the foundation for all parallelism, concurrency, and asynchrony in .NET. According to Microsoft, `Task` is the only preferred way to write multithreaded and parallel code [26].

2.4 Parallel

The `Parallel` class is a part of the TPL library. It provides three main methods to support parallel programming design patterns: data parallelism (via `Parallel.For` and `Parallel.ForEach`), and task parallelism (via `Parallel.Invoke`).

`Parallel.For` method accepts three parameters: an inclusive lower-bound, an exclusive upper-bound, and a lambda function to be invoked for each iteration. By default, it uses the work queued to .NET thread pool to execute the loop with as much parallelism as it can muster. `Parallel.For(0, n, (i)=> foo(i));`

`Parallel.ForEach` is a very specialized loop. Its purpose is to iterate through a specific kind of data set, a data set made up of numbers that represent a range. `Parallel.ForEach(books, (book)=>foo(book))`

`Parallel.Invoke` runs the operations (lambda functions) given as parameters concurrently and waits until they are done. It parallelizes the operations, not the data. `Parallel.Invoke(()=> foo(), ()=> boo());`

`Parallel` class works efficiently even if developers pass in an array of one million lambda functions to `Parallel.Invoke` or one million iterations to `Parallel.For`. This is because `Parallel` class does not necessarily use one `Task` per iteration or operation, as that could add significantly more overhead than is necessary. Instead, it partitions the large number of input elements into batches and then it assigns each batch to a handful of underlying tasks. Under the covers, it tries to use the minimum number of tasks necessary to complete the loop (for `For` and `ForEach`) or operations (for `Invoke`) as fast as possible. Hence, Microsoft shows that `Parallel` class performs faster than equivalent `Task`-based code in some cases [15].

`Parallel` class will run iterations or operations in parallel unless this is more expensive than running them sequentially. The runtime system handles all thread scheduling details, including scaling automatically to the number of cores on the host computer.

3 Motivation

Before explaining TASKIFIER and SIMPLIFIER, we explore the motivations of these tools by answering two research questions:

Q1: What level of parallel abstractions do developers use?
Q2: What do developers think about parallel abstractions?

We first explain how we gather the code corpus to answer these questions. We use the same code corpus to evaluate our tools (Section 6).

3.1 Methodology

We created a code corpus of C# apps by using our tool COLLECTOR. We chose GitHub [12] as the source of the code corpus because Github is now the most popular open-source software repository, having surpassed Google Code and SourceForge.

COLLECTOR downloaded the most popular 1000 C# apps which have been modified at least once since June 2013. COLLECTOR visited each project file in apps in order to resolve/install dependencies by using nuget [17], the package manager of choice for apps targeting .NET. COLLECTOR also eliminated the apps that do not compile due to missing libraries, incorrect configurations, etc. COLLECTOR made as many projects compilable as possible (i.e., by resolving/installing dependencies).

COLLECTOR also eliminated 72 apps that targeted old platforms (e.g., Windows Phone 7, .NET Framework 3.5, Silverlight 4) because these old platforms do not support new parallel libraries.

After all, COLLECTOR successfully retained 880 apps, comprising 42M SLOC, produced by 1859 developers. This is the corpus that we used in our analysis and evaluation.

In terms of the application domain, the code corpus has (1) 364 libraries or apps for desktops, (2) 185 portable-libraries for cross-platform development, (3) 137 Windows Phone 8 apps, (4) 84 web apps (ASP.NET), (5) 56 tablet applications (Surface WinRT), and (6) 54 Silverlight apps (i.e., client-side runtime environment like Adobe Flash). Hence, the code corpus has apps which (i) span a wide domain and (ii) are developed by different teams with 1859 contributors from a large and varied community.

Roslyn: The Microsoft Visual Studio team has released Roslyn [22] with the goal to expose compiler-as-a-service through APIs to other tools like code generation, analysis, and refactoring. Roslyn has components such as Syntax, Symbol Table, Binding, and Flow Analysis APIs. We used these APIs in our tools for analyzing our code corpus.

Roslyn also provides the Services API allowing to extend Visual Studio. Developers can customize and develop IntelliSense, refactorings, and code formatting features. We used Services API for implementing our tools.

3.2 Q1: What level of parallel abstractions do developers use?

In a previous study [18], we found out that developers prefer to use old style threading code over `Task` in C# apps. We wanted to have a newer code corpus which includes the recently updated most popular apps. We used Roslyn API to get the usage statistics of the abstractions.

As we explained in Section 2, there are 4 main ways to offload a computation to another thread: (1) creating a `Thread`, (2) accessing the `ThreadPool` directly, (3) creating a `Task`, (4) using task or data parallelism patterns with `Parallel.Invoke` and `Parallel.For(Each)`. Table 1 tabulates the usage statistics of all these approaches. Some apps use more than one parallel idiom and some never use any parallel idiom.

Table 1 Usage of parallel idioms. The three columns show the total number of abstraction instances, the total number of apps with instances of the abstraction, and the percentage of apps with instances of the abstraction.

	#	App	App%
Creating a `Thread`	2105	269	31%
Using `ThreadPool`	1244	191	22%
Creating a `Task`	1542	170	19%
Data Parallelism Pattern with `Parallel.For(Each)`	432	51	6%
Task Parallelism Pattern with `Parallel.Invoke`	53	12	1%

As we see from the table, developers use `Thread` and `ThreadPool` more than `Task` and `Parallel` even though our code corpus contains recently updated apps which target the latest versions of various platforms. The usage statistics of `Parallel` are also very low compared to `Task`. These findings definitely show that developers use low-level parallel abstractions.

Surprisingly, we also found that 96 apps use `Thread`, `ThreadPool`, and `Task` at the same time. This can easily confuse the developer about the scheduling behavior.

3.3 Q2: What do developers think about parallel abstractions?

In this question, we explore why developers use low-level abstractions and whether they are aware of the newer abstractions.

We first asked the experts on parallel programming in C#. We looked for the experts on StackOverflow [20] which is the pioneering Q&A website for programming. We contacted the top 10 users for the tags “multithreading" and “C#", and got replies from 7 of them. Among them are Joe Albahari who is the author of several books on C# (e.g., “C# in a Nutshell"), and John Skeet who is the author of “C# in Depth" and he is regarded as one of the most influential people on StackOverflow.

All of them agree that `Task` should be the only way for parallel and concurrent programming in C#. For example, one said “*Tasks should be the only construct*

for building multithreaded and asynchronous applications". According to them, `Thread` should be used for testing purposes: "*threads are actually useful for debugging*" (e.g., guaranteeing a multithreading environment, giving names to threads). When we asked them whether an automated tool is needed to convert `Thread` to `Task`, they concluded that the existence of some challenges makes the automation really hard. For example, one said that "*I wonder whether doing it nicely in an automated fashion is even feasible*" and another said that "*Often there's in-brain baggage about what the thread is really doing which could affect what the target of the refactoring should actually be*".

Second, we contacted the developers of 10 applications which heavily mix `Thread`, `ThreadPool`, and `Task`. Most of them said that the legacy code uses `Thread` and `ThreadPool` and they always prefer `Task` in the recent code. The developer of the popular ravendb application [1], Oren Eini, said that "*We intend to move most stuff to tasks, but that is on an as needed basis, since the code works*" and another said that his team "*never had time to change them*". This comment indicates that the changes are tedious.

We also asked the developers whether they are aware of the `Parallel` class. Developers of 7 of the apps said that they are not aware of the `Parallel` class and they were surprised seeing how much it decreases the code complexity: "*Is this in .NET framework? It is the most elegant way of a parallel loop*".

4 Taskifier

We developed Taskifier, a tool that migrates `Thread` and `ThreadPool` abstractions to `Task` abstractions. Section 4.1 presents the algorithms for the migration from `Thread` to `Task`. Section 4.2 presents the migration from `ThreadPool` to `Task`. Section 4.3 presents the special cases to handle some challenges. Section 4.4 presents how developers interact with Taskifier.

4.1 Thread to Task

First, Taskifier needs to identify the `Thread` instances that serve as the target of the transformation. In order to do this, Taskifier detects all variable declarations of `Thread` type (this also includes arrays and collections of `Thread`). For each `Thread` variable, it iterates over its method calls (e.g., `thread.Start()`) and member accesses (e.g., `thread.IsAlive=...`). Then, Taskifier replaces each of them with their correspondent from the `Task` class. However, corresponding operations do not necessarily use the same name. For instance, `thread.ThreadState`, an instance field of `Thread` class gets the status of the current thread. The same goal is achieved in `Task` class by using `task.Status`.

Some low-level operations in `Thread` do not have a correspondent in the `Task` class. For example, (1) Priority, (2) Dedicated Name, (3) Apartment State.

After studying both `Thread` and `Task`, we came up with a mapping between them. Taskifier uses this map for the conversion. If Taskifier finds operations that have no equivalents, it will discard the whole conversion from `Thread` to `Task` for that specific `Thread` variable.

The most important transformations in the uses of `Thread` variables are for creating, starting, and waiting operations. Code list. 1.5 shows a basic usage of `Thread` and Code list. 1.6 represents the equivalent code with `Task` operations. Developers create `Thread` by using its constructor and providing the asynchronous computation. There are various ways of specifying the computation in the constructor such as delegates, lambdas, and method names. In the example below, a delegate (`ThreadStart`) is used. TASKIFIER gets the computation from the delegate constructor and transforms it to a lambda function. For starting the `Thread` and `Task`, the operation is the same and for waiting, `Task` uses `Wait` instead of `Join`.

Code 1.5 Simple Thread example

```
ThreadStart t = new ThreadStart(doWork);
Thread thread = new Thread(t);
thread.Start();
thread.Join();
```

Code 1.6 Equivalent Task code

```
Task task = new Task(()=>doWork());
task.Start();
task.Wait();
```

While the transformation in Code listings 1.5 and 1.6 shows the most basic case when the asynchronous computation does not take any arguments, the transformation is more involved when the computation needs arguments. Consider the example in Code listing 1.7. The asynchronous computation is the one provided by the `Reset` method (passed in line 1), but the parameter of the `Reset` method is passed as an argument to the `Thread.Start` in line 3. Since the `Thread.Start` can only take `Object` arguments, the developer has to downcast from `Object` to a specific type (in line 7).

Code listing 1.8 shows the refactored version, that uses `Task`. Unlike in `Thread`, `Task.Start` does not take a parameter. In order to pass the state argument `e` to the asynchronous computation `Reset`, the code uses a lambda parameter in the `Task` constructor. In this case, since there is no need to cast parameters in the `Reset` method body, TASKIFIER also eliminates the casting statement (Line 7 from Code list. 1.7).

Code 1.7 Thread with dependent operators from Dynamo [3] app

```
1  ParameterizedThreadStart threadStart = new ParameterizedThreadStart(Reset);
2  Thread workerThread = new Thread(threadStart);
3  workerThread.Start(e);
4  ...
5  private void Reset(object state)
6  {
7    var args = (MouseButtonEventArgs)state;
8    OnClick(this, args);
9    ...
10 }
```

⇓

Code 1.8 Code listing 1.7 migrated to Task

```
1  Task workerTask = new Task(()=>Reset(e));
2  workerTask.Start();
3  ...
4  private void Reset(MouseButtonEventArgs args)
5  {
6    OnClick(this, args);
7    ...
8  }
```

TASKIFIER also changes the variable names such as from `workerThread` to `workerTask` by using the built-in Rename refactoring of Visual Studio.

After TASKIFIER migrates the `Thread` variable to `Task`, it makes an overall pass over the code again to find some optimizations. For instance, in Code listing 1.8, there is no statement between `Task` constructor and `Start` method. In `Task`, there is a method combining these two statements: `Task.Run` creates a `Task`, starts running it, and returns a reference to it. TASKIFIER replaces the first two lines of Code listing 1.8 with only one statement: `Task workerTask = Task.Run(()=>Reset(e));`

TASKIFIER successfully detects all variable declarations of `Thread` class type; however, we noticed that developers can use threads through an anonymous instance. The example below from antlrcs app [2] shows such an anonymous usage of `Thread` on the left-hand side, and refactored version with `Task` on the right-hand side. TASKIFIER replaces the `Thread` constructor and the start operation with a static method of `Task`.

```
new Thread(t1.Run).Start(arg);      =>    Task.Run(()=>t1.Run(arg));
```

4.2 ThreadPool to Task

The conversion from `ThreadPool` to `Task` is less complex than the previous transformation. There is only one static method that needs to be replaced, `ThreadPool.QueueUserWorkItem(...)`. TASKIFIER simply replaces this method with the static `Task.Run` method and removes the parameter casting from `Object` to actual type in the beginning of the computation. The example below illustrates the transformation.

```
WaitCallback operation= new WaitCallback(doSendPhoto);
ThreadPool.QueueUserWorkItem(operation, e);
```

```
Task.Run(()=>DoSendPhoto(e));
```

4.3 Special Cases

There are three special cases that make it non-trivial to migrate from `Thread` and `ThreadPool` to `Task` manually:

1. I/O or CPU-bound Thread: During manual migration, developers need to understand whether the candidate thread for migration is I/O or CPU bound since it can significantly affect performance. If an I/O-bound `Thread` is transformed to a `Task` without special consideration, it can cause starvation for other tasks in the thread pool. Some blocking synchronization abstractions like `Thread.Sleep` can also cause starvation when the delay is long.

Manually determining whether the code in a `Thread` transitively calls some blocking operations is non-trivial. It requires deep inter-procedural analysis.

When developers convert `Thread` to `Task` manually, it is easy to miss such blocking operations that appear deep inside the methods called indirectly from the body of the `Thread`. In our code corpus, we found that 32% of tasks have at least one I/O blocking operation and 9% use `Thread.Sleep` that blocks the thread longer than 1 second. It shows that developers are not aware of this issue and their tasks can starve.

Thus, it is crucial for Taskifier to determine whether the nature of the computation is I/O or CPU-bound. If it finds blocking calls, it converts them into non-blocking calls, in order to avoid starvation.

To do so, Taskifier checks each method call in the call graph of the `Thread` body for a blocking I/O operation by using a blacklist approach. For this check, we have the list of all blocking I/O operations in .NET. If Taskifier finds a method call to a blocking I/O operation, it tries to find an asynchronous (non-blocking) version of it. For example, if it comes across a `stream.Read()` method call, Taskifier checks the members of the `Stream` class to see if there is a corresponding `ReadAsync` method. Upon finding such an equivalent, it gets the same parameters from the blocking version. `ReadAsync` is now non-blocking and returns a future `Task` to get the result when it is available. After finding the corresponding non-blocking operation, Taskifier simply replaces the invocation with the new operation and makes it `await`'ed. When a `Task` is awaited in an `await` expression, the current method is paused and control is returned to the caller. The caller is the thread pool so the thread pool will choose another task instead of busy-waiting. When the await'ed `Task`'s background operation is completed, the method is resumed from right after the `await` expression.

```
var string = stream.Read();      =>      var string = await stream.ReadAsync();
```

If Taskifier cannot find asynchronous versions for all blocking I/O operations in the `Thread` body, it does not take any risks of blocking the current thread and, instead, it inserts a flag to the `Task` creation statement: `TaskCreationOptions.LongRunning`. This flag forces the creation of a new thread outside the pool. This has the same behavior as the original code, i.e., it explicitly create a new `Thread`. But now the code still enjoys the many other benefits of using Tasks, such as compatibility with the newer libraries and brevity.

In the case of `Thread.Sleep`, Taskifier replaces this blocking operation with a timer-based non-blocking version, `await Task.Delay`. Upon seeing this statement, the thread in the thread pool does not continue executing its task and another task from the thread pool is chosen (cooperative-blocking).

2. Foreground and Background Thread: By default, a `Thread` runs in the foreground, whereas threads from `ThreadPool` and `Task` run in the background. Background threads are identical to foreground threads with one exception: a background thread does not keep the managed execution environment running. `Thread` is created on the foreground by default but can be made background by

"`thread.IsBackground = true`" statement. If a developer wants to execute `Task` in a foreground thread, she has to add some extra-code in the body of `Task`.

Since the intention is to preserve the original behavior as much as possible, TASKIFIER should do the transformations accordingly. In the example below, the program will not terminate until the method, `LongRunning` reaches the end. However, when this `Thread` is turned into `Task` without any special consideration, the program will not wait for this method and it will immediately terminate. While it is easy to diagnose the problem in this simple example, it can be really hard for a fairly complex app.

```
public static void main(String args[])
{
  ...
  new Thread(LongRunning);
}
```

Although, in some cases, TASKIFIER is able to tell from the context if the thread is foreground or background, it is usually hard to tell if the developer really intended to create a foreground thread. Developers usually do not put much thought into a thread's being a foreground thread when created. We chose to implement our algorithm for TASKIFIER to transform `Thread` to `Task` by default to work in the background. The developer still has the option of telling TASKIFIER to create foreground tasks; however, the reasoning behind going with the background by default is that when we contacted the developers, most of them did not want the `Task` to work in the foreground even though they created foreground threads.

3. Exception Handling: Another difference between `Thread` and `Task` is the mechanism of unhandled exceptions. An unhandled exception in `Thread` and `ThreadPool` abstractions results in termination of the application. However, unhandled exceptions that are thrown by user code that is running inside `Task` abstractions are propagated back to the joining thread when the static or instance `Task.Wait` methods are used. For a thrown exception to be effective in a `Task`, that `Task` should be waited; otherwise, the exceptions will not cause the termination of the process.

A simple direct migration from `Thread` and `ThreadPool` to `Task` can make the unhanded exceptions silenced so developers will not notice them. This situation may destroy the reliability and error-recovery mechanism that developers put into the original program.

To take care of this, TASKIFIER adds a method call to make sure exception handling is preserved and unhandled exceptions are not ignored when non-waited threads are migrated to tasks. During the transformation of the example below, TASKIFIER adds a new method, `FailFastOnException` to the project just once. Other instances of `Task` in the project can use this method. However, this stage is optional and can be enabled by the user upon request.

```
new Thread(method).Start();
void method()
{
  throw new Exception();
}
```

```
Task.Run(()=>method()).FailFastOnException();
void method()
{
  throw new Exception();
}
public static Task FailFastOnException(this Task task)
{
    task.ContinueWith(c => Environment.FailFast("Task faulted", c.Exception),
        TaskContinuationOptions.OnlyOnFaulted |
        TaskContinuationOptions.ExecuteSynchronously |
        TaskContinuationOptions.DetachedFromParent);
    return task;
}
```

4.4 Workflow

We implemented Taskifier as a Visual Studio plugin, on top of the Roslyn SDK [22]. Because developers need to run Taskifier only once per migration, Taskifier operates in a batch mode. The batch option allows the programmer to migrate automatically by selecting any file or project in the IDE. Before starting the migration, Taskifier asks the user for two preferences: *Foreground Thread* option and *Exception Handling* option. When it operates at the file levels, Taskifier might still modify other files when necessary (e.g., if the method in `Thread` body is located in another file). Taskifier migrates `Thread` and `ThreadPool` abstractions to `Task` in about 10 seconds on an average project (100K SLOC).

5 Simplifier

Taskifier automatically migrates old-style parallel abstractions (`Thread` and `ThreadPool`) to the modern `Task`. However, there are still some opportunities for higher-level abstractions that can make the code faster and more readable.

`Parallel` class (see Section 2.4) provides parallel programming design patterns as a higher-level abstraction over `Task` class. Implementing these design patterns with tasks requires developers to write code with several instances of Tasks. A much simpler alternative is to use a single instance of the `Parallel` class, which encapsulates the main skeleton of the design patterns. While the direct usage of Tasks affords more flexibility and control, we found out that in many cases, developers do not use the extra flexibility, and their code can be greatly simplified with a higher-level design pattern.

We developed Simplifier that converts multiple `Task` instances to one of three `Parallel` operations (`Parallel.For`, `Parallel.ForEach`, `Parallel.Invoke`). Simplifier suggests code snippets that can be transformed to `Parallel` operations and then does the actual transformation on demand. Hence, we divided the explanation of the algorithms into two parts: *Suggestion* and *Transformation*. In the Suggestion part, we explain how Simplifier chooses the code candidates. In the Transformation part, we explain how Simplifier transforms these candidates to `Parallel` operations. After explaining the three algorithms, we discuss how developers interact with Simplifier in Section 5.4.

5.1 Multiple Tasks to Parallel.Invoke

Simplifier offers the transformation of task parallelism pattern composed of a group of `Task` instances to `Parallel.Invoke`. First we explain the properties of code snippets that can be transformed to this operation.

Suggestion: As we explained in Section 2.4, `Parallel.Invoke` is a succinct way of creating and starting multiples tasks and waiting for them. Consider the example below. `Parallel.Invoke` code on the right-hand side is the equivalent of the code on the left-hand side. For the purpose of simplifying the code with `Parallel.Invoke`, Simplifier needs to detect such a pattern before suggesting a transformation.

Code 1.9 Multiple Tasks

```
Task t1=new Task(()=>sendMsg(arg1));
Task t2=new Task(()=>sendMsg(arg2));
t1.Start();
t2.Start();
Task.WaitAll(t1,t2);
```

Code 1.10 Equivalent with Invoke

```
Parallel.Invoke(()=>sendMsg(arg1),
                ()=>sendMsg(arg2));
```

Listing 1.9 shows the simplest form of many variations of code snippets. In order to find as many fits as possible, we need to relax and expand this pattern to detect candidates. First step to detect the pattern is that the number of `Task` variables should be at least 2, as `Parallel.Invoke` can take unlimited work items as parameters. Second, Simplifier has to consider that there are many syntactic variations of task creation and task starting operations. Also, there are some operations that combine both *creation* and *starting* like `Task.Factory.StartNew` and `Task.Run` methods. Third, one should keep in mind that there is no need to separate the creation of Tasks into one phase and starting them into another. Each Task can be created then started immediately. Fourth, there may be other statements executing concurrently in between the start of a `Task` and the barrier instruction that waits for all spawned tasks. In case of such statements, Simplifier encapsulates them in another `Task` and passes the task to `Parallel.Invoke`. Code listing 1.11 shows a more complex pattern of task parallelism from a real-world app and demonstrates the last point.

After Simplifier finds out the code snippets that fit into the pattern stated above, it checks if some preconditions hold true to ensure that the transformation is safe. These preconditions are not limitations of Simplifier; they are caused by how `Parallel.Invoke` encapsulates the task parallelism pattern. Because it is a higher-level abstraction, it waives some advanced features of `Task`. The preconditions are:

P1: None of the `Task` variables in the pattern can be result-bearing computations, i.e., a *future* – `Task<ResultType>` – also called a *promise* in C#. The reason is that after the transformation, there is no way to access the result-bearing from the `Parallel` class.

P2: There should be no reference to the `Task` variables outside of code snippet of the design pattern. Such references will no longer bind to a `Task` after the transformation eliminates the `Task` instances.

P3: None of the `Task` variables in the pattern can use the chaining operation (`ContinueWith`). Since the chaining requires access to the original task, this task will no longer exist after the transformation.

Transformation: If SIMPLIFIER finds a good match of code snippets, its suggestion can be executed and turned into a transformation which yields `Parallel.Invoke` code. Code listing 1.12 shows the code after the transformation of Code listing 1.11.

During transformation, the main operation is to get work items from `Task` variables. In the example below, the work item of first `Task` is `()=> DoClone(...)`. These work items can be in different forms such as method identifiers, delegates, or lambda functions as in the example below. SIMPLIFIER handles this variety of forms by transforming the work items to lambda functions.

After SIMPLIFIER gets the work items for the tasks `t1` and `t2`, it forms another work item to encapsulate the statements between task creation and task waiting statements (line 3 and 4 in Code List. 1.11).

SIMPLIFIER gives all these work items in the form of lambda functions to `Parallel.Invoke` method as parameters. It replaces the original lower-level task parallelism statements with this `Parallel.Invoke` method.

Code 1.11 Candidate from Kudu [5] app

```
1 var t1 = Task.Factory.StartNew(() => DoClone("PClone1", appManager));
2 var t2 = Task.Factory.StartNew(() => DoClone("PClone2", appManager));
3 ParseTheManager();
4 DoClone("PClone3", appManager);
5 Task.WaitAll(t1, t2);
```

Code 1.12 Equivalent Parallel.Invoke code

```
1 Parallel.Invoke(() => DoClone("PClone1", appManager),
2                 () => DoClone("PClone2", appManager),
3                 () => {ParseTheManager();
4                        DoClone("PClone3", appManager);})
```

5.2 Tasks in Loop to Parallel.For

SIMPLIFIER can transform a specific data parallelism pattern to `Parallel.For`. First we explain the properties of code snippets that can be transformed to this operation.

Suggestion: As we explained in Section 2.4, `Parallel.For` is a more concise way to express the pattern of forking several tasks and then waiting for them all to finish at a global barrier.

Considering the example below, the `Parallel.For` code on the right is the equivalent of the code on the left.

Code 1.13 Forking tasks in a loop

```
Task[] tasks = new Task [n];
for(int i=0; i<n; i++)
{
  int temp = i;
  tasks[i]= new Task(
              ()=>Queues[temp].Stop());
  tasks[i].Start();
}
Task.WaitAll(tasks);
```

Code 1.14 Equivalent with Parallel.For

```
Parallel.For(0,n,(i)=> Queues[i].Stop());
```

Simplifier needs to detect usages of Tasks that form the pattern on the left example above. The code snippet in Listing 1.13 is one of the basic representatives of this design pattern; there are other variations who fit the pattern. First thing the tool looks for in the code to decide if it matches the pattern is that the increment operation of the loop must be of the form `++` or `+= 1` (i.e., increments should only be by 1). The loop boundaries do not matter as long as they are integers. Second, as explained in Sec. 5.1, there may be many syntactic variations for the task creation and starting operations.

Third, the collection of tasks does not have to be of type `Array`, they may be of another type like `List`. In this case, tasks are added with `tasks.add(...)` method to the collection in the loop. Fourth, as long as there is no modification to the collection, there may be other statements between creating the collection of tasks and the for loop. During the transformation, these statements are not discarded and they take place before `Parallel.For`.

Fifth, there might be other statements in the loop besides task creation, starting, and adding to the collection. In the Code List. 1.13 above, there is one such statement: `int temp=i;`. This causes each task to have its own copy of the loop index variable during the iteration of the loop.

Sixth and last, some simple assignment operations may also exist between the loop and the barrier operation that waits for all spawned tasks. Code listing 1.15 shows a more complex pattern of data parallelism from a real-world app and demonstrates the last point with the statements in Line 8-9.

After Simplifier detects the code snippets that fit into the pattern stated above, it checks some preconditions ensuring that the transformation is safe. These preconditions are the result of how the `Parallel.For` encapsulates the date parallelism pattern.

P1, P2, P3: The first three preconditions are the same as the first three preconditions in Sec. 5.1.

P4: The operations in the loop except the task-related statements should not carry any dependence between iterations. Consider the Code listing 1.15, the statements in Line 4-5 will sequentially execute because they are not included in `Task`. After transforming to `Parallel.For`, the whole body of the loop will be parallelized.

P5: The statements after the loop (e.g., Line 8-9 in Code List. 1.15) but before the `Task.WaitAll` should not access any data from the Task body. At first, Simplifier did not allow any statement between the loop and `Task.WaitAll`. After we manually analyzed the statements between the loop and `WaitAll` in

our code corpus, we noticed that many of them are simple variable declarations which do not use any data from the loop and do not contain any method call sites like in the Code List. 1.15. Therefore, we relaxed this precondition and allowed the statements after the loop but before the `Task.WaitAll` unless they do not access any data from the `Task` body in the loop. To detect such cases, SIMPLIFIER used an intra-procedural data-flow analysis to determine that these statements are independent from the loop. Roslyn [22] provides ready-to-use control & data flow analysis APIs that SIMPLIFIER used to understand how variables flow in and out of regions of source.

Transformation: Code listing 1.16 shows the code after the transformation of Code listing 1.15 showing a more complex example.

During transformation, the main operation is to the get loop boundaries and the work item from the task in the loop. In the example below (Line 6), the work item is `()=> MultiSearcherCallableNoSort(...)`. The loop boundaries are 0 and `tasks.Length`. However, the collection of tasks will be deleted after the transformation. Hence, when SIMPLIFIER detects such a dependence on the size of task collections in the loop boundaries, it replaces this boundary with the original size of the task collection, which is `searchables.Length`.

Then, SIMPLIFIER needs to make sure that the statements in the loop (e.g. Line 4-5 in List. 1.15) are not dependent on loop iterations. If they are not, these statements are put in the beginning of the work item; otherwise, the transformation will not occur. If one of these statements is the temporary holder of the iteration value like `cur = i` in the example below, SIMPLIFIER removes it and replaces the holder (`cur`) with the iteration variable (`i`) in the work item as seen in List. 1.16.

Lastly, SIMPLIFIER replaces the original lower-level data parallelism statements with the `Parallel.For` method.

Code 1.15 Candidate from lucene.net [6] app

```
1  Task[] tasks = new Task[searchables.Length];
2  for (int i = 0; i < tasks.Length; i++)
3  {
4    int cur = i;
5    cur = callableIterate(cur);
6    tasks[i] = Task.Factory.StartNew(() => MultiSearcherCallableNoSort(cur, ...));
7  }
8  int totalHits = 0;
9  float maxScore = float.NegativeInfinity;
10 Task.WaitAll(tasks);
```

⇓

Code 1.16 Equivalent Parallel.For code

```
1 Parallel.For(0, searchables.Length, (i) => {
2                             i = callableIterate(i);
3                             MultiSearcherCallableNoSort(i, ...);} );
4 int totalHits = 0;
5 float maxScore = float.NegativeInfinity;
```

5.3 Tasks in Loop to Parallel.ForEach

While this transformation is very similar to `Parallel.For`, it transforms `foreach` loops instead of `for` loops. `foreach` loops are a special case of loops that are used to iterate over the elements of a collection.

First we explain the properties of code snippets that can be transformed to this operation.

Suggestion: Considering the example below, the `Parallel.ForEach` code on the right is the equivalent of the code on the left.

Code 1.17 Equivalent Task example

```
Task[] tasks = new Task[sables.Length];
foreach (var sable in sables)
{
  tasks[i] = Task.Run(
          () => sable.DocFreq(term));
}
Task.WaitAll(tasks);
```

Code 1.18 Parallel.ForEach example

```
Parallel.ForEach(sables,
                 (sable)=>sable.DocFreq(term));
```

Simplifier needs to detect usages of Tasks in a `foreach` loop that form the pattern on the left example above. We will generalize this pattern with the same 5 variations in the `Parallel.For` algorithm, except the first one which represents the custom loop boundaries.

After Simplifier detects the code snippets that fit into the pattern, it checks for the same preconditions as in the `Parallel.For` transformation.

Transformation: Code listing 1.20 shows the code after the transformation of Code listing 1.19. The transformation is done in a very similar manner with the `Parallel.For` version, except the loop boundaries.

After the work item is extracted from `Task`, Simplifier needs to get the collection variable and iteration variable from the loop declaration (`functions`, `functionText`). Then, Simplifier replaces the original lower-level data parallelism statements with the `Parallel.ForEach` method.

Code 1.19 Candidate from Jace [4] app

```
1  List<Task> tasks = new List<Task>();
2  foreach (string functionText in functions)
3  {
4    Task task = new Task(() =>
5    {...
6      function(functionText, ...); ...
7    });
8    tasks.Add(task);
9    task.Start();
10 }
11 Task.WaitAll(tasks.ToArray());
```

⇓

Code 1.20 Equivalent Parallel.ForEach code

```
1 Parallel.ForEach(functions,(functionText) =>{...
2                                    function(functionText, ...); ...
3                                  });
```

5.4 Workflow

We implemented SIMPLIFIER as a Visual Studio plugin, on top of the Roslyn SDK [22]. SIMPLIFIER's workflow is similar to a "quick hint" option which exists in major IDEs such as Eclipse, Netbeans, IntelliJ. SIMPLIFIER scans the file that is open in the editor in real-time. It tries to find code snippets that fit into the patterns of the three transformations discussed above. Because it executes on the background (triggered by any keystroke), the analysis of finding code snippets should be fast enough to prevent sluggishness. However, the analyses for `Parallel.For(Each)` require some expensive checking of preconditions such as P4 and P5 in Sec. 5.2. Because they require dependence and data-flow analyses, we do not execute them in the suggestion phase, but in the transformation phase.

If SIMPLIFIER finds candidates, it suggests the places where the transformations can be useful by underlining the code snippet and displaying a hint in the sidebar. After the user clicks the hint and confirms, SIMPLIFIER transforms the code for the `Parallel.Invoke`. SIMPLIFIER tests long-running preconditions, such as for the `Parallel.For(Each)`, in the transformation phase. If the candidate passes these preconditions too, the code will be transformed to the `Parallel.For(Each)`. If not, SIMPLIFIER will give an informative warning.

6 Evaluation

We conducted two kinds of empirical evaluation. First, we *quantitatively* evaluate based on case studies of using our tools on open-source software. Second, we *qualitatively* evaluate based on patches that we sent to open-source developers.

6.1 Quantitative

To quantitatively evaluate the usefulness of TASKIFIER and SIMPLIFIER, we answer the following research questions:

RQ1: How **applicable** are the tools?
RQ2: Do the tools reduce the **code bloat**?
RQ3: How much **programmer effort** is saved by the tools?
RQ4: Are the automated transformations **safe**?

Experimental Setup: To answer the questions above, we ran TASKIFIER and SIMPLIFIER on our code corpus that we gathered from Github. The code corpus has 880 C# apps, comprising 42M SLOC, spanning a wide spectrum from web & desktop apps to libraries and mobile apps.

We ran both tools in batch mode over this code corpus. Even though SIMPLIFIER was not designed to run in a batch mode, we implemented a batch mode specifically for the purpose of the evaluation. TASKIFIER visits all `Thread` variable declarations and anonymous instances, and applies the migration algorithm. SIMPLIFIER finds the candidates of code snippets for each source file, then transforms the snippets to the targeted pattern.

Table 2 summarizes the results for the first three research questions.

Table 2 Taskifier and Simplifer Conversion Results. The first column shows the total number of instances that the tool applied. The second column shows the total number of instances that the tool successfully converted and the third column shows the percentage of successfully transformed instances. The fourth column shows the total number of reduced SLOC by the transformations and the fifth column shows the percentage of the reduced lines. The last column shows the total number of modified SLOC.

	Applicability			Reduction		Modified
	Applied	Conv.	Conv. %	SLOC	%.	SLOC
`Thread` to `Task`	1782	1390	78%	2244	24%	8876
`ThreadPool` to `Task`	1244	1244	100%	173	14%	2115
`Task` to `Parallel.Invoke`	85	85	100%	502	44%	1870
`Task` to `Parallel.For(Each)`	205	188	92%	1918	62%	5640

RQ1: How applicable are the tools? Out of our corpus of 880 apps, 269 used Threads (see Table 1). Together, they account for 2105 `Thread` instances. Based on our discussion with experts (see Section 3.3), they suggested we discard `Thread` usages in test code because developers may need threads for enforcing a multithreading testing environment. After eliminating the `Thread` usages in test code, we were left with 1782 `Thread` instances in production code, as shown in Table 2.

Taskifier migrated 78% of the `Thread` instances. The remaining 22% of `Thread` instances used operations that are not available in the `Task` class, thus are not amenable for migration. For example, one can set up the name of a Thread, but not of a Task. Deciding whether the name is important requires domain knowledge, thus Taskifier stays on the safe side and warns the programmer.

Because there are no preconditions for the migration of `ThreadPool` instances, Taskifier migrated all of them to `Task`.

As for Simplifier, it successfully transformed 100% of the 85 `Task`-based fork-join patterns to `Parallel.Invoke`. Out of the 205 identified `Task`-based data-parallelism patterns, it transformed 92% to `Parallel.For` or `Parallel.ForEach`. The remaining 8% did not pass the preconditions. A major number of them was failed due to P4: loop-carried dependence.

RQ2: Do the tools reduce the code bloat? The second column, *Reduction*, of Table 2 shows by how much each tool eliminates bloated code. As we expect, because Simplifier transforms *multiple* `Task` operations and helper operations to *one* equivalent method in the `Parallel` class (i.e., a *many-to-one* transformation), it has the largest impact. For the transformation to `Parallel.Invoke`, Simplifier achieved on average a 44% reduction in SLOC for each code snippet that it transformed. For the transformation to `Parallel.For(Each)`, it achieved on average a 62% reduction for each transformed code snippet.

Taskifier migrates *one* `Thread` operation to *one* equivalent `Task` operation (i.e., a *one-to-one* transformation), so we expect modest reductions in LOC. These

come from optimizations such as combining the creation and start `Task` operations, removing explicit casting statements which are not needed in `Task` bodies, etc. However, the advantages brought by TASKIFIER are (i) the modernization of the legacy code so that it can now be used with the newer platforms, and (ii) the transformation of blocking operations to the equivalent non-blocking operations.

RQ3: How much programmer effort is saved by the tools? The last column of Table 2 shows that the transformations are tedious. Had the programmers manually changed the code, they would have had to manually modify 10991 SLOC for the migration to `Task` and 7510 SLOC for the migration to `Parallel`.

Moreover, these changes are non-trivial. TASKIFIER found that 37% of `Thread` instances had at least one I/O blocking operation. To find these I/O blocking operations, TASKIFIER had to check deeper in the call-graphs of `Thread` bodies, which span 3.4 files on average. SIMPLIFIER found that 42% of the loops it tried to transform contained statements that needed an analysis to identify loop-carried dependences.

RQ4: Are the automated transformations safe? We used two means to check the safety of our transformations. First, after our tools applied any transformation, our evaluation script compiled the app in-memory and determined that no compilation errors were introduced. Second, we sampled and manually checked 10% of all transformed instances and determined that they were correct. Also, the original developers of the source code thought that the transformations were correct (see Section 6.2).

In contrast to the code that was transformed with the tools, we found that 32% of the `Task`-code manually written by open-source developers contained at least one I/O blocking operation which can cause serious performance issues (see Section 4.3). However, the code transformed by TASKIFIER into `Task` instances does not have this problem.

6.2 Qualitative evaluation

To further evaluate the usefulness our of tools in practice, we identified actively developed C# applications, we ran our tools on them, and submitted patches [3] to the developers.

For TASKIFIER, we selected the 10 most recently updated apps that use `Thread` and `ThreadPool` and transformed them with TASKIFIER. We submitted 52 patches via a pull request. Developers of 8 apps out of 10 responded, and accepted 42 patches.

We received very positive feedback on these pull requests. Some developers said that migration to `Task` is on their TODO list but they always postponed it because of working on new features. It is tedious to migrate `Task` and developers

[3] All patches can be found on our web site: `Taskifier.NET`

can easily miss some important issues such as blocking I/O operations during the migration. TASKIFIER helps them migrate their code in a fast and safe manner.

For SIMPLIFIER, we selected a different set of 10 most recently updated apps that had a high chance of including good matches of code snippets for `Parallel.For(Each)` or `Parallel.Invoke` patterns. We submitted 14 patches. Developers of 7 apps out of 10 responded, and accepted 11 patches. All of them liked the new code after the transformation and asked us whether we can make the tool available now.

6.3 Discussion

As explained in Section 4.3, TASKIFIER analyzed the call graph of `Thread` body to detect I/O blocking operations, using a blacklist approach. Although we have the list of I/O blocking operations in .NET framework, TASKIFIER is not aware of I/O blocking operations implemented by 3rd-party libraries whose source code is not available in the app. However, we don't expect that the number of blocking I/O operations implemented by external libraries to be high.

Most of non-blocking I/O and synchronization operations were released in .NET 4.5 (2012). If an application does not target .NET 4.5, it cannot take advantage of the non-blocking operations. However, applications that are targeting the new platforms (e.g, Windows Phone 8, Surface) are forced to use .NET 4.5.

With respect to releasing TASKIFIER and SIMPLIFIER, we will be able to publish the tools when Microsoft publicly releases the new version of Roslyn (expected by Spring '14). Because we used an internal version of Roslyn, we had to sign an NDA, which prohibits us from releasing tools based on Roslyn.

7 Related Work

Empirical Studies about Parallelism Pankratius et al. [21] analyzed concurrency related transformations in a few Java applications. Torres et al. [25] conducted a study on the usage of concurrent programming constructs in Java, by analyzing around 2000 applications. In our previous study [10] we cataloged the kinds of changes that Java developers perform when they write parallel code. We have also conducted an empirical study [18] on how developers from thousands of open source projects use C# parallel libraries.

In this paper, we do not only target the usage statistics of parallel abstractions but also provides TASKIFIER and SIMPLIFIER for helping developers migrate from low-level parallel abstractions to higher-level abstractions.

Refactoring Tools for Parallelism: There are a few refactoring tools that specifically target concurrency. Dig et al. [9,11] retrofit parallelism into sequential applications via Java concurrent libraries. In the same spirit, Wloka et al. [27] present a refactoring for replacing global state with thread local state. Schafer et al. [24] present Relocker, a refactoring tool that lets programmers replace usages of Java built-in locks with more flexible locks. Schafer et al. [23] also investigated the

problem of whether existing sequential refactorings are concurrency-aware. Gyori et al. [13] present Lambdaficator, that refactors existing Java code to use lambda expressions to enable parallelism. We previously studied asynchronous programming and developed Asyncifier [19], an automated refactoring tool that converts old style asynchronous code to use new language features (i.e., `async/await`). However, none of these previous tools address the problem of migrating between different levels of abstractions in (already) parallel code.

Balaban et al. [8] present a tool for converting between obsolete classes and their modern replacements. The developer specifies a mapping between the old APIs and the new APIs. Then, the tool uses a type-constraint analysis to determine if it can replace all usages of the obsolete class. Their tool supports a one-to-one transformation whereas SIMPLIFIER supports many-to-one transformations. Even our one-to-one transformations from TASKIFIER require custom program analysis, e.g., detecting I/O blocking operations, and cannot be simply converted by a mapping program.

8 Conclusions

To make existing parallel code readable, faster, and scalable, it is essential to use higher-level parallel abstractions. Their usage is encouraged by the industry leaders as the old, low-level abstractions are subject to deprecation and removal in new platforms.

Our motivational study of a corpus of 880 C# applications revealed that many developers still use the lower-level parallel abstractions and some are not even aware of the better abstractions. This suggests a new workflow for transformation tools, where suggestions can make developers aware of new abstractions.

Converting from low-level to high-level abstractions can not be done by a simple find-and-replace tool, but it requires custom program analysis and transformation. For example, 37% of `Thread` instances use blocking I/O operations, which need special treatment when they are converted to `Task` instances, otherwise it can create severe performance bugs. We found that 32% instances of manually written `Task` indeed contain blocking I/O operations.

In this paper we presented two tools. Our first tool, TASKIFIER, converts `Thread`-based usage to lightweight `Task`. We were surprised that despite some differences between `Thread` and `Task` abstractions, 78% of the code that uses `Thread` can be successfully converted to `Task`. Our second tool, SIMPLIFIER, converts `Task`-based code into higher-level parallel design patterns. Such conversions reduce the code bloat by 57%. The developers of the open-source projects accepted 53 of our patches and are looking forward to using our tools.

Acknowledgements: This research is partly funded through NSFCCF-1439957 and CCF-1442157 grants, a SEIF award from Microsoft, and a gift grant from Intel. The authors would like to thank Cosmin Radoi, Yu Lin, Mihai Codoban, Caius Brindescu, Sergey Shmarkatyuk, Alex Gyori, Michael Hilton, and anonymous reviewers for providing helpful feedback on earlier drafts of this paper.

References

1. RavenDB 2nd generation document database. May '14, `http://ravendb.net`.
2. Antlrcs. May '14, `http://github.com/antlr/antlrcs`.
3. Dynamo App. May '14, `https://github.com/ikeough/Dynamo`.
4. Jace App. May '14, `https://github.com/pieterderycke/Jace`.
5. Kudu App. May '14, `https://github.com/projectkudu/kudu`.
6. Lucene.NET App. May '14, `https://github.com/apache/lucene.net`.
7. Tiraggo App. May '14, `https://github.com/BrewDawg/Tiraggo`.
8. Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the OOPSLA '05*, pages 265–279, 2005.
9. Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proceedings of the ICSE '09*, pages 397–407, 2009.
10. Danny Dig, John Marrero, and Michael D. Ernst. How do programs become more concurrent: A story of program transformations. In *Proceedings of the IWMSE '11*, pages 43–50, 2011.
11. Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. Relooper. In *Proceedings of the OOPSLA '09*, pages 793–794, 2009.
12. Github. May '14, `https://github.com`.
13. Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the FSE '13*, pages 543–553, 2013.
14. Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.
15. Daan Leijen and Judd Hall. Parallel Performance: Optimize Managed Code For Multi-Core Machines. *MSDN*, October 2007.
16. Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *ACM SIGPLAN Notices*, 44(10):227, October 2009.
17. Nuget. May '14, `http://www.nuget.org/`.
18. Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the FSE '12*, pages 54–65, 2012.
19. Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in C# . In *Proceedings of the ICSE '14*, pages 1117–1127, 2014.
20. Stack Overflow. May '14, `http://stackoverflow.com`.
21. Victor Pankratius, Christoph Schaefer, Ali Jannesari, and Walter F. Tichy. Software engineering for multicore systems. In *Proceedings of the IWMSE'08*, pages 53–60, 2008.
22. The Roslyn Project. May '14, `http://msdn.microsoft.com/en-us/hh500769`.
23. Max Schäfer, Julian Dolby, Manu Sridharan, Emina Torlak, and Frank Tip. Correct refactoring of concurrent java code. In *Proceedings of the ECOOP '10*, pages 225–249, 2010.
24. Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java programs for flexible locking. In *Proceedings of the ICSE '11*, pages 71–80, 2011.
25. Weslley Torres, Gustavo Pinto, Benito Fernandes, João Paulo Oliveira, Filipe Alencar Ximenes, and Fernando Castor. Are Java programmers transitioning to multicore?: a large scale study of java FLOSS. In *Proceedings of the SPLASH '11 Workshops*, pages 123–128, 2011.
26. Steven Toub. Patterns of Parallel Programming. *Microsoft Corporation*, 2010.
27. Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the FSE '09*, pages 173–182, 2009.